PHP Mapping for Classes

Previous Next

On this page:

Basic PHP Mapping for Classes

A Slice class maps to a PHP class with the same name. For each Slice data member, the generated class contains a member variable, just as for structures and exceptions. Consider the following class definition:

Slice

```
class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
}
```

The PHP mapping generates the following code for this definition:

PHP

```
class TimeOfDay extends \Ice\Value
{
    public function __construct($hour=0, $minute=0, $second=0)
    {
        $this->hour = $hour;
        $this->minute = $minute;
        $this->second = $second;
    }

    public function ice_id()
    {
        return '::TimeOfDay';
    }

    public static function ice_staticId()
    {
        return '::TimeOfDay';
    }

    public function __toString()
    {
        // ...
    }

    public $hour;
    public $minute;
    public $second;
}
```

There are a number of things to note about the generated code:

1. The generated class `TimeOfDay` inherits from `\Ice\Value`. This reflects the semantics of Slice classes in that all classes implicitly inherit from `\Ice\Value`, which is the ultimate ancestor of all classes. Note that `\Ice\Value` is *not* the same as `\Ice\ObjectPrx`. In other words, you *cannot* pass a class where a proxy is expected and vice versa.
2. The constructor initializes an instance variable for each Slice data member.
3. The class defines the method `ice_id` and class method `ice_staticId`.

There is quite a bit to discuss here, so we will look at each item in turn.

Back to Top ^

Inheritance from `Value` in PHP

Like interfaces, classes implicitly inherit from a common base class, `\Ice\Value`. However, classes inherit from `\Ice\Value` instead of `\Ice\ObjectPrx`, therefore you cannot pass a class where a proxy is expected (and vice versa) because the base types for classes and proxies are not compatible.

`Value` defines the following functions:

PHP

```
class Value
{
    public function ice_id()
    {
        return "::Ice::Object";
    }

    public function ice_preMarshal()
    {
    }

    public function ice_postMarshal()
    {
    }

        public function ice_getSlicedData()
    {
        ...
    }

    public static function ice_staticId()
    {
        return "::Ice::Object";
    }
}
```

These functions behave as follows:

- `ice_id`
  This method returns the actual run-time type ID of the object. If you call `ice_id` through a reference to a base instance, the returned type ID is the actual (possibly more derived) type ID of the instance.

- `ice_preMarshal`
  If the object defines this method, the Ice run time invokes it just prior to marshaling the object's state, providing the opportunity for the object to validate its declared data members.

- `ice_postUnmarshal`
  If the object defines this method, the Ice run time invokes it after unmarshaling the object's state. An object typically defines this method when it needs to perform additional initialization using the values of its declared data members.
- `ice_getSlicedData`
  This functions returns the `SlicedData` object if the value has been sliced during un-marshaling or `null` otherwise.

- `ice_staticId`
  This method is generated in each class and returns the static type ID of the class.

Back to Top ^

Class Data Members in PHP

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding member variable.

Optional data members use the same mapping as required data members, but an optional data member can also be set to the marker value `\Ice\Unset` to indicate that the member is unset. A well-behaved program must compare an optional data member to `Unset` before using the member's value:

PHP

```
$v = ...;
if($v->optionalMember == \Ice\None)
{
    echo "optionalMember is unset\n";
}
else
{
    echo "optionalMember = " . $v->optionalMember . "\n";
}
```

The `Unset` marker value has different semantics than `null`. Since `null` is a legal value for certain Slice types, the Ice run time requires a separate marker value so that it can determine whether an optional value is set. An optional value set to `null` is considered to be set. If you need to distinguish between an unset value and a value set to `null`, you can do so as follows:

PHP

```
$v = ...;
if($v->optionalMember == \Ice\None)
{
    echo "optionalMember is unset\n";
}
else if($v->optionalMember == null)
{
    echo "optionalMember is null\n";
}
else
{
    echo "optionalMember = " . $v->optionalMember . "\n";
}
```

If you wish to restrict access to a data member, you can modify its visibility using the `protected` metadata directive. The presence of this directive causes the Slice compiler to generate the data member with protected visibility. As a result, the member can be accessed only by the class itself or by one of its subclasses. For example, the `TimeOfDay` class shown below has the `protected` metadata directive applied to each of its data members:

Slice

```
class TimeOfDay
{
    ["protected"] short hour;   // 0 - 23
    ["protected"] short minute; // 0 - 59
    ["protected"] short second; // 0 - 59
    string format();    // Return time as hh:mm:ss
}
```

The Slice compiler produces the following generated code for this definition:

PHP

```
abstract class TimeOfDay extends \Ice\Value
{
    public function __construct($hour=0, $minute=0, $second=0)
    {
        $this->hour = $hour;
        $this->minute = $minute;
        $this->second = $second;
    }
```

```
    public function ice_id()
    {
        return '::TimeOfDay';
    }

    public static function ice_staticId()
    {
        return '::TimeOfDay';
    }

    public function __toString()
    {
        // ...
    }

    protected $hour;
    protected $minute;
    protected $second;
}
```

For a class in which all of the data members are protected, the metadata directive can be applied to the class itself rather than to each member individually. For example, we can rewrite the `TimeOfDay` class as follows:

Slice

```
["protected"] class TimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
}
```

Back to Top ^

Class Constructors in PHP

Classes have a constructor that assigns to each data member a default value appropriate for its type:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	Null
`dictionary`	Null
`class/interface`	Null

You can also declare different default values for data members of primitive and enumerated types.

For derived classes, the constructor has one parameter for each of the base class's data members, plus one parameter for each of the derived class's data members, in base-to-derived order.

Pass the marker value `\Ice\None` as the value of any optional data members that you wish to be unset.

Back to Top ^

Class Operations in PHP

Deprecated Feature

Operations on classes are deprecated as of Ice 3.7. Skip this section unless you need to communicate with old applications that rely on this feature.

With the PHP mapping, operations in classes are not mapped at all into the corresponding PHP class. The generated PHP class is the same whether the Slice class has operations or not.

Value Factories in PHP

While value factories were necessary in previous versions of Ice when using classes with operations (a now deprecated feature) with the PHP mapping, value factories may be used for any kind of class and are *not* deprecated.

Value factories allow you to create classes derived from the PHP class generated by the Slice compiler, and tell the Ice run time to create instances of these classes when unmarshaling. For example, with the following simple interface:

Slice

```
class CustomTimeOfDay extends TimeOfDay
{
    public function format() { ... prints formatted data members ... }
}
```

You then create and register a value factory for your custom class with your Ice communicator:

PHP

```
class ValueFactory implements \Ice\ValueFactory
{
    public function create($type)
    {
        if($type == TimeOfDay::ice_staticId()))
        {
            return new CustomTimeOfDay;
        }
        assert(false);
        return null;
    }
}

$communicator->getValueFactoryManager()->add(new ValueFactory(), TimeOfDay::ice_staticId())
```

Back to Top ^

See Also

- Classes
- Type IDs
- Optional Data Members
- Value Factories

Previous Next